Chapitre 7

Programmation modulaire

Un long programme est difficile à appréhender globalement. Il vaut donc mieux le scinder en petits programmes : un **programme principal** fait appel à des **sous-programmes**, qui peuvent eux-mêmes faire appel à des sous-programmes, du programme principal ou de celui-ci, et ainsi de suite. C'est le principe du **raffinement successif**. De plus certains sous-programmes peuvent servir dans d'autres programmes, c'est le principe de la **modularité**. Ces principes sont mis en œuvre en langage C grâce aux *fonctions*.

On peut distinguer, en langage C, les **fonctions prédéfinies** des bibliothèques (telles que `printf()` ou `scanf()`), livrées avec le compilateur et « intégrées » au programme lors de l'édition des liens, et les fonctions que le programmeur écrit lui-même en tant que partie du texte source. Nous avons déjà vu comment utiliser les premières. Nous allons donc nous intéresser ici aux secondes. Nous verrons aussi, d'ailleurs, la façon de concevoir les fonctions prédéfinies.

7.1 Un exemple

Introduction.- Commençons par donner un exemple simple de définition et d'utilisation d'une fonction.

Considérons à nouveau, pour cela, notre exemple de fonction que l'on veut évaluer en un certain nombre de points. Une nouvelle amélioration consiste à dégager la définition de la fonction en utilisant un sous-programme.

Programme.- La mise en place suivante est intuitivement compréhensible, nous la détaillerons ensuite :

```
/* Fonct_1.c */
#include <stdio.h>
#include <math.h>

double f(double x)
       {
       return((sin(x) + log(x))/(exp(x) + 2));
       }

void main(void)
     {
     float x, y;
     printf("x = ");
     scanf("%f",&x);
     while (x != 1000)
       {
       y = f(x);
       printf("f(%f) = %f\n", x, y);
       printf("x = ");
       scanf("%f",&x);
       }
     }
```

L'amélioration provient ici plus particulièrement du fait que l'on ne s'occupe pas de la fonction particulière dans le corps du programme, mais uniquement du fait que l'on veut afficher sa valeur en un certain nombre de points, ce qui est l'essence du programme. Il suffit de changer le sous-programme, bien mis en évidence, lorsqu'on veut changer de fonction.

7.2 Définition d'une fonction

La définition des fonctions, dans des cas simples, est suffisamment claire au vu du programme précédent. Mais l'explicitation suivante donne les limites d'une utilisation intuitive.

7.2.1 Syntaxe

7.2.1.1 Règles de définition

Nom d'une fonction.- Une fonction a un **nom** qui est un identificateur (non utilisé pour une autre entité, comme d'habitude).

Syntaxe de la définition.- Une fonction se définit. La syntaxe de la définition est la suivante :

```
type nom(type1 variable1, ..., typen variablen)
{
corps de la fonction
}
```

la première ligne étant l'**en-tête** de la fonction, le reste le **corps** de la fonction.

Le corps de la fonction est constitué exactement comme dans le cas de la fonction principale `main()`, la seule considérée jusqu'à maintenant.

Sémantique.- Ceci correspond à la définition d'une fonction (à plusieurs variables) définie sur une partie de $A_1 \times A_2 \times ... \times A_n$ et à valeurs dans B, où A_1 est l'ensemble des entités du type `type1`, ... , A_n l'ensemble des entités du type `typen` et B l'ensemble des entités du type `type`.

Le type vide.- Il est prévu un type lorsque la fonction ne renvoie pas de valeur, ou lorsqu'elle n'a pas d'argument, le type `void`. Nous avons déjà rencontré ce type depuis le début.

7.2.1.2 Emplacement des définitions

Un programme C est une suite de définitions de fonctions, la définition de la fonction `main()` étant obligatoire :

```
fonction1
fonction2
.........
fonctionn
```

Une définition de fonction ne doit pas contenir de définitions de fonctions (contrairement à ce qui se passe dans d'autres langages de programmation, par exemple PASCAL).

Il faut que les fonctions soient définies avant d'être appelées (pour qu'elles soient connues du compilateur). Nous verrons comment faire lorsque ce n'est pas le cas.

Le programme commence par exécuter la fonction `main()` et ne se sert des autres fonctions que si il est fait appel à elles, soit directement dans la fonction principale, soit indirectement *via* d'autres fonctions appelées.

7.2.1.3 Valeur de retour

Notion.- Le type précédant le nom de la fonction est celui de la valeur de la fonction, appelée **valeur de retour** en langage C.

Type de la valeur de retour.- La valeur de retour d'une fonction peut être soit d'un type de base (`char`, `short`, `int`, `long`, `float`, `double`), soit un pointeur (notion que nous verrons plus tard) vers n'importe quel type (simple ou complexe) de données.

L'instruction de renvoi.- Les valeurs de retour des fonctions sont renvoyées à la fonction appelante grâce à l'instruction `return`.

Emplacement des instructions de retour.- La rencontre (au cours de l'exécution, pas de la définition) d'une instruction `return` met fin à l'exécution des instructions d'une fonction et rend le contrôle du programme à la fonction appelante.

Les parenthèses ne sont pas obligatoires. On peut écrire :

```
return(expression);
```

ou :

```
return expression;
```

Si le type de `expression` ne coïncide pas avec le type de la valeur retournée tel qu'il est défini dans l'en-tête de la fonction, alors le compilateur le convertit comme il faut (avec tous les aléas possibles).

La spécification de `expression` est optionnelle. Si elle est omise alors la valeur retournée est indéfinie.

7.2.1.4 Paramètres

Notion.- Les arguments d'une fonction sont appelés ses **paramètres** en langage C. Le nom de la fonction est suivi de la liste des paramètres et de leurs types.

Paramètres formels et paramètres effectifs.- Il faut bien faire la différence entre les paramètres au moment de la définition d'une fonction et les paramètres qui seront donnés lors de l'**appel** d'une fonction.

Les paramètres spécifiés lors de la définition d'une fonction sont qualifiés de **paramètres formels**. Les paramètres transmis à la fonction lors de son appel de la fonction sont appelés les **paramètres effectifs**.

Un paramètre formel est une variable locale, connue seulement à l'intérieur de la fonction pour laquelle est définie ce paramètre.

Les paramètres formels et effectifs doivent correspondre en nombre et en type ; par contre les noms pouvent différer.

7.2.1.5 Appel d'une fonction

L'utilisation d'une fonction (on parle d'**appel** de fonction en langage C) se fait exactement comme en Mathématiques. Soit une fonction de nom `f`, d'un type de retour réel et dont les paramètres sont, dans cet ordre, `x`, `y` et `z` respectivement de type, disons, entier, réel et entier. Si `m`, `x` et `n` sont des expressions de types respectifs entier, réel et entier alors `f(m,x,n)` est une expression de type réel.

7.2.2 Exemples de définition

Introduction.- Donnons quelques exemples variés de définitions pour illustrer ce que nous venons de dire.

Exemple 1.- (**La fonction sinc**)

La fonction `f` définie par l'expression $f(x) = \frac{sinx}{x}$ est *a priori* définie sur $\mathbb{R}^*$ mais on sait la prolonger par continuité en 0 en lui attribuant la valeur 1. Ceci permet de définir la fonction suivante :

```
double f(double x)
   {
   double y;
   if (x == 0) y = 1;
   else y = sin(x)/x;
   return(y);
   }
```

ou

```
double f(double x)
   {
   if (x == 0) return(1);
   else return(sin(x)/x));
   }
```

Ceci nous donne un exemple avec deux utilisations de `return`, à cause de la définition par cas.

Exemple 2.- (**L'exponentiation**)

Programmons l'exponentiation, c'est-à-dire l'application de $\mathbb{R}\times\mathbb{N}$ dans $\mathbb{R}$ qui au couple (x, n) associe x^n :

```
double puiss(double x, int n)
   {
   double y;
   int i;
   y = 1;
   for (i=1; i <= n; i++) y = y*x;
   return(y);
   }
```

Cet exemple est intéressant à plusieurs titres : la fonction a deux paramètres et, de plus, de types différents ; le corps comporte des déclarations de variables, dites **variables locales** ; on utilise une boucle pour définir la fonction. On remarquera que nous sommes obligés d'introduire la variable locale `y` car `puiss` ne pourrait pas être utilisé à droite d'une affectation.

Exemple 3.- (**Fonction sans paramètre**)

Considérons le programme suivant :

```
/* intro.c */
#include <stdio.h>
```

```
void welcome(void)
     {
     printf("Welcome to this fantastic program");
     printf("\nwhich proves to you the power");
     printf("\nof the modularity. \n");
     }

void bienvenue(void)
     {
     printf("Bienvenue dans ce merveilleux");
     printf("\nprogramme qui vous montre la");
     printf("\npuissance de la modularite. \n");
     }

void main(void)
     {
     char c;
     printf("Do you want to continue in");
     printf("\nEnglish or in French (E/F) ");
     scanf("%c",&c);
     if (c == 'E') welcome();
     else bienvenue();
     }
```

Les fonctions `welcome()` et `bienvenue()` n'ont ni paramètre ni valeur de retour. Remarquons cependant l'utilisation du type `void` et le couple de parenthèses pour l'appel de ces fonctions. Le fait qu'il n'y ait pas de type de retour rend l'utilisation de `return` superflue.

Exemple 4.- (**Le pgcd**)

Nous avons déjà rencontré en exercice le calcul du pgcd de deux entiers non nuls, ce qui donnait lieu à l'amélioration successive d'algorithmes pour le calculer. Le programme suivant repose sur l'*algorithme d'Euclide*.

```
int pgcd(int a, int b)
   {
   int r;
   r = 1;
   while (r != 0)
      {
      r = a % b;
      a = b;
      b = r;
      }
   return(a);
   }
```

Cet exemple est intéressant pour deux raisons : la fonction a deux paramètres de même type et l'algorithme est non trivial.

7.2.3 Un exemple d'utilisation : la dichotomie

Considérons la méthode de calcul approché de la solution d'une équation par la **méthode de dichotomie**. Rappelons tout d'abord en quoi cela consiste.

7.2.3.1 Rappel

Principe.- Soit f une fonction réelle de la variable réelle, définie sur un segment $[a, b]$, continue, strictement monotone et telle que $f(a)$ et $f(b)$ soient de signes opposés. Alors il existe une et une seule valeur c de $[a, b]$ telle que $f(c) = 0$.

Même si on sait calculer $f(x)$ pour toute valeur x de $[a, b]$, il n'est pas évident de trouver une **valeur exacte** de c. Par contre pour tout $\epsilon > 0$ on peut trouver une valeur $\bar{c}$ telle que : $|\bar{c} - c| < \epsilon$, c'est-à-dire une **valeur approchée** de c à ϵ près.

On divise l'intervalle en deux parties égales en posant $d = \frac{a+b}{2}$: si $f(d)$ est de même signe que $f(a)$ alors c appartient à l'intervalle $[d, b]$, sinon il appartient à l'intervalle $[a, d]$. On peut donc restreindre l'intervalle de recherche en posant, par exemple dans le premier cas, $a = d$ et $b = b$. Le nouvel intervalle est d'amplitude moitié. On recommence ainsi jusqu'à ce que l'amplitude soit inférieure à ϵ et on peut alors prendre pour $\bar{c}$ n'importe quelle valeur du dernier intervalle $[a, b]$, par exemple a.

Exercice.- *Montrer que l'équation* $x^5 - x + 1 = 0$ *vérifie bien les conditions d'application de la méthode de dichotomie sur l'intervalle* $[-2, -1]$.

7.2.3.2 Programme sans fonction

Un programme.- Le programme suivant permet de mettre en œuvre la méthode de dichotomie, illustrée par l'exemple précédent, sans utiliser la notion de fonction :

```
/* dicho_1.c */

#include <stdio.h>
#include <math.h>

void main(void)
     {
     float a, b, d, e, ya, yb, yd;
     printf("Dichotomie \n");
     printf("a = ");
     scanf("%f", &a);
     printf("b = ");
     scanf("%f", &b);
     printf("erreur < ");
     scanf("%e", &e);
     ya = a*a*a*a*a - a + 1;
     yb = b*b*b*b*b - b + 1;
     while (b - a >= e)
        {
        d = (a + b)/2;
        yd = d*d*d*d*d - d + 1;
        if (ya*yd < 0)
           {
```

```
            b = d;
            yb = yd;
            }
         else
            {
            a = d;
            ya = yd;
            }
         }
      printf("c = %e. \n", d);
     }
```

Si on fait exécuter ce programme avec les valeurs :
$a = -2$, $b = -1$, $erreur < 1.0e-6$,
on obtient : $c = -1.167304e+00$.

7.2.3.3 Programme avec fonction

Un deuxième programme.- Un programme utilisant la notion de fonction est le suivant :

```
/* dicho_2.c */

#include <stdio.h>
#include <math.h>

float f(float x)
       {
       return(x*x*x*x*x - x + 1);
       }

void main(void)
      {
      float a, b, d, e;
      printf("Dichotomie \n");
      printf("a = ");
      scanf("%f", &a);
      printf("b = ");
      scanf("%f", &b);
      printf("erreur < ");
      scanf("%e", &e);
      while (b - a >= e)
         {
         d = (a + b)/2;
         if  ( f(a)*f(d) < 0 ) b = d;
         else a = d;
         }
      printf("c = %e\n", d);
      }
```

Si on fait exécuter ce programme, on obtient évidemment le même résultat qu'avec le programme précédent. Mais ce programme est plus clair et plus modulaire (lorsqu'on veut changer de fonction,

il suffit de changer le corps de la fonction).

Exercice.- En fait la présentation du résultat n'est pas satisfaisante : on préférerait six chiffres après la virgule correspondant aux six premiers chiffres du développement décimal de c.

Concevoir un programme qui demande, non pas l'erreur, mais le nombre de chiffres après la virgule que l'on désire et qui donne le résultat expliqué dans la phrase précédente.

Ne pas oublier que le tronqué d'une valeur approchée ne convient pas toujours : par exemple $1,979$ est une valeur approchée à 10^{-2} de $1,986$ mais le résultat escompté est $1,98$ et non $1,97$.

7.3 Mode de transmission des paramètres

7.3.1 Notion

Introduction.- Pour une fonction mathématique f à deux arguments, lorsqu'on évalue cette fonction, en effectuant par exemple $y = f(a, b)$, on s'attend à ce que la valeur de y soit changée mais pas les valeurs de a ou de b. En informatique il en va autrement. On peut, par exemple, désirer écrire un sous-programme :

```
echange(a,b)
```

dont le but est d'échanger les valeurs de a et de b. Nous savons écrire le corps de ce programme en langage C depuis longtemps. Par exemple si a et b sont des entiers on a :

```
int c;
c = a;
a = b;
b = c;
```

Comment écrire le sous-programme correspondant dans un langage de programmation ?

Passage par valeur et passage par référence.- Lorsqu'on ne veut pas changer la valeur d'un paramètre on parle de **transmission par valeur** (en anglais **call by value**), c'est-à-dire que le sous-programme ne reçoit qu'une copie de la valeur. Lorsqu'on veut se permettre la possibilité de changer le valeur on parle de **transmission par référence** (en anglais **call by reference**).

Mise en place.- Beaucoup de langages permettent ces deux types de transmission. C'est le cas, par exemple, du langage PASCAL. Le langage C, quant à lui, ne connaît que la transmission par valeur, la transmission par référence est émulée en envoyant la valeur de l'adresse grâce aux pointeurs (comme nous le verrons ci-dessous).

7.3.2 Passage par valeur

Introduction.- Voyons sur deux exemples le comportement de la transmission par valeur.

Exemple 1.- Considérons le programme suivant :

```
/* valeur_1.c */

#include <stdio.h>

void affiche(int n)
     {
```

```
     printf("%d \n", n);
     }

void main(void)
     {
     int n = 3;
     affiche(n);
     }
```

Son exécution fait afficher 3 comme attendu.

Exemple 2.- Considérons le programme modifié suivant :

```
/* valeur_2.c */

#include <stdio.h>

void affiche(int n)
     {
     n = 4;
     printf("%d \n", n);
     }

void main(void)
     {
     int n = 3;
     affiche(n);
     printf("%d \n", n);
     }
```

Son exécution fait afficher 4 puis 3. La fonction `affiche()` n'a donc pas changé la valeur de `n` dans la fonction principale.

7.3.3 Variable globale

Introduction.- Une **variable globale** est une variable déclarée avant toute définition de fonction. L'intérêt est qu'elle est alors connue de toutes les fonctions, on peut donc changer sa valeur dans chacune d'elle. L'inconvénient est qu'on peut aussi changer sa valeur par inadvertance. Il vaut donc mieux utiliser le passage par référence qu'une variable globale. Voyons cependant comment cela fonctionne.

Exemple 1.- Considérons le programme suivant :

```
/* global_1.c */

#include <stdio.h>

int n = 3;

void affiche(void)
     {
```

```
     n = 4;
     printf("%d \n", n);
     }

void main(void)
     {
     affiche();
     printf("%d \n", n);
     }
```

L'exécution de ce programme fait afficher 4 puis 4. La fonction `affiche()` a donc changé la valeur de `n` dans la fonction principale.

Remarque.- L'utilisation des variables globales dans un programme est permise, mais il vaut mieux éviter de s'en servir. Cela va en effet à l'encontre de la clarté de la modularité. Lorsqu'on considère un sous-programme on devrait voir d'un seul coup d'oeil toutes les variables qui y interviennent. C'est tout l'intérêt des arguments.

Exemple 2.- Remarquons que, dans le cas de la variable globale ci-dessus, la fonction `affiche()` n'avait pas d'argument. Considérons, par opposition, le programme suivant :

```
/* global_2.c */

#include <stdio.h>

int n = 3;

void affiche(int n)
     {
     n = 4;
     printf("%d \n", n);
     }

void main(void)
     {
     affiche(n);
     printf("%d \n", n);
     }
```

Son exécution fait afficher 4 puis 3. La déclaration de l'argument `n` dans la fonction `affiche()` le fait considérer comme une **variable locale**. La variable `n` du corps de cette fonction correspond à la dernière variable déclarée, donc à la variable locale, et non à la variable globale ; ceci explique pourquoi la valeur de la variable globale, elle, n'a pas changée.

7.3.4 Variable locale

Introduction.- La procédure elle-même peut avoir besoin de variables qui n'ont pas d'intérêt pour le programme en son entier, comme nous l'avons vu à propos de la fonction `puiss()`. On parle alors de **variable locale**.

Remarque.- Le nom de *variable locale* se justifie de par la position de sa déclaration. Elle se justifie aussi car une telle variable n'est pas connue du programme dans son entier mais uniquement de la fonction (plus généralement du bloc) dans laquelle elle a été déclarée.

Ceci montre l'intérêt d'utiliser le plus possible des variables locales : c'est une protection supplémentaire contre les erreurs de programmation.

Nom d'une variable locale.- Nous avons dit précédemment (avant de considérer les fonctions) qu'on ne peut pas déclarer le même identificateur comme variable à deux endroits différents. Il faut assouplir un petit peu cette règle avec l'utilisation des sous-programmes : en effet, pour un (gros) programme, des sous-programmes différents peuvent être conçus par des personnes différentes, qui peuvent donc penser au même nom de variable ; ceci sera détecté lors de la compilation, certes, mais cela risque de ne pas faciliter les choses. On permet donc en général, dans un langage de programmation, la possibilité d'utiliser un même identificateur plusieurs fois comme variable locale.

Priorité des variables locales sur les variables globales.- Un problème se pose alors :

Quelle est la variable considérée lors de son utilisation ?

La réponse est simple : celle qui a été déclarée le plus récemment, c'est-à-dire la plus locale à l'intérieur du sous-programme (et, plus généralement, à l'intérieur du bloc).

7.4 Déclaration des fonctions

7.4.1 Notion

Nous avons dit qu'un programme C est une suite de définitions de fonctions. Cette façon de faire risque de donner lieu à des programmes confus lorsqu'on a besoin d'un millier de fonctions, par exemple. Il n'y a en général aucune raison que la 522-ième fonction fasse directement appel aux 521 fonctions précédentes. On distingue donc les notions de *déclaration* et de *définition* d'une fonction. Nous avons déjà vu la notion de définition ; venons-en à celle de déclaration.

DÉFINITION.- *Une* **déclaration de fonction** *fournit des indications sur le nom de la fonction, sur le type de la valeur retournée et sur le type des paramètres.*

Il faut, avec cette nouvelle notion, revenir sur la règle disant qu'une fonction doit être définie avant d'être utilisée. La règle est en fait la suivante.

RÈGLE 1.- *Une fonction doit être déclarée ou définie avant d'être utilisée. Elle doit être déclarée dans un module dans lequel elle est utilisée si elle est définie dans un autre.*

Nous reviendrons plus tard sur la deuxième partie de la règle. Indiquons aussi une autre règle.

RÈGLE 2.- *Une fonction peut être déclarée plusieurs fois, mais elle ne peut être définie qu'une seule fois.*

7.4.2 Syntaxe de la déclaration de fonction en C

La déclaration de fonction en langage C peut se faire suivant deux modes, appelés **prototypes**.

Prototype complet.- *La syntaxe de la déclaration d'une fonction sous forme de* **prototype complet** *est la suivante :*

```
type nom(type1 variable1, ... , typen variablen);
```

où `type, type1, ... ,typen` *sont des types et où* `nom` *et* `variable1, ... , variablen` *sont des identificateurs (le nom de la fonction et les* **noms des paramètres formels***).*

Prototype simplifié.- *La syntaxe de la déclaration d'une fonction sous forme de* **prototype simplifié** *est la suivante* :

```
type nom(type1, ... , typen);
```

Autrement dit, on renonce aux noms des paramètres, puisqu'ils ne sont pas utiles.

Situation de la déclaration.- La **déclaration locale** d'une fonction se fait au début du bloc de la définition d'une fonction, avant les déclarations de variables. La fonction ainsi déclarée n'est connue que de la fonction dans laquelle elle est déclarée. On peut aussi utiliser une **déclaration globale** d'une fonction avant toute définition de fonction.

Exemple de déclaration locale.- Le programme suivant illustre la notion de déclaration locale de fonction :

```
/* dec_loc.c */
#include <stdio.h>
#include <math.h>

void main(void)
     {
     double f(double x);
     float x, y;
     printf("x = ");
     scanf("%f",&x);
     while (x != 1000)
       {
       y = f(x);
       printf("f(%f) = %f\n", x, y);
       printf("x = ");
       scanf("%f",&x);
       }
     }

double f(double x)
     {
     return((sin(x) + log(x))/(exp(x) + 2));
     }
```

Exemple de déclaration globale.- Le programme suivant illustre la notion de déclaration globale de fonction :

```
/* dec_glo.c */
#include <stdio.h>
#include <math.h>
```

```
double f(double x);

void main(void)
     {
     float x, y;
     printf("x = ");
     scanf("%f",&x);
     while (x != 1000)
       {
       y = f(x);
       printf("f(%f) = %f\n", x, y);
       printf("x = ");
       scanf("%f",&x);
       }
     }

double f(double x)
       {
       return((sin(x) + log(x))/(exp(x) + 2));
       }
```

7.4.3 Pragmatique

Pour mettre un peu d'ordre dans un programme faisant intervenir beaucoup de fonctions, on peut commencer par la définition de la fonction principale, précédée de la déclaration des fonctions dont elle a besoin. On définit ensuite ces fonctions auxiliaires, chacune d'elles précédée des déclarations des fonctions auxiliaires de second niveau dont elles ont besoin, et ainsi de suite.

7.5 Déclaration de fonctions dans des fichiers en-tête

Introduction.- La déclaration d'une fonction personnelle ne doit pas nécessairement être incluse explicitement dans le même fichier du programme source que la fonction principale. Elle peut aussi se trouver dans un fichier en-tête, comme pour les fonctions prédéfinies. Ce dernier est ensuite inséré dans le programme grâce à une commande du préprocesseur.

Exemple.- En mettant dans le même répertoire les deux fichiers suivants :

```
/* calcul.c */
#include <stdio.h>
#include <math.h>
#include "defini.h"

void main(void)
     {
     float x, y;
     printf("x = ");
     scanf("%f",&x);
     while (x != 1000)
        {
        y = f(x);
```

```
        printf("f(%f) = %f\n", x, y);
        printf("x = ");
        scanf("%f",&x);
        }
    }
```

et :

```
/* defini.h */

double f(double x)
       {
       return((sin(x) + log(x))/(exp(x) + 2));
       }
```

on obtient un programme correct qui fonctionne.

Commentaires.- 1^o) Le fichier en-tête `definiti.h` contient ici non pas la déclaration d'une fonction mais sa définition. On place en principe le code des fonctions directement dans le programme ou dans des bibliothèques. Les définitions des fonctions livrées avec le compilateur se trouvent (sous forme de code objet) dans de telles bibliothèques ayant l'extension `.lib` (pour *LIBrary*).

2^o) Le fichier en-tête est écrit entre guillemets verticaux $''$ et non entre les symboles $<$ et $>$ car il ne se trouve pas dans le répertoire standard des en-têtes mais dans le répertoire contenant le programme source.

On peut aussi indiquer le chemin complet, par exemple en MS-DOS (Windows) :

```
#include "c:\tcwin\programmes\definiti.h"
```

7.6 Les pointeurs et le passage par adresse

Nous avons vu l'intérêt d'utiliser des arguments. L'utilisation des arguments transmis par valeur évite bien des erreurs. Il existe cependant des cas où l'on veut réellement changer la valeur de l'argument en appelant une fonction, c'est-à-dire effectuer une *transmission par référence*. Ce mode de passage des arguments n'a pas été prévu en langage C, ce qui est un défaut. On peut cependant, bien sûr, l'émuler : ceci se fait grâce à la *transmission par adresse*. Pour cela nous avons besoin de connaître comment manipuler les adresses des variables grâce à la notion de *pointeur*.

Les *pointeurs* jouent très tôt un rôle important en langage C (ce qui est un défaut), parce que l'adressage indirect est important pour le passage par adresse des fonctions. C'est ce premier aspect que nous allons voir ici ; nous reviendrons plus tard sur les pointeurs lors de la définition des *structures de données dynamiques*.

7.6.1 Les pointeurs et l'adressage indirect

Commençons par voir la notion d'*adressage indirect*, mise en place à l'aide des *pointeurs*, notion qui n'a rien à voir avec les fonctions *a priori*.

7.6.1.1 Notion générale

Principe.- Jusqu'à maintenant nous avons vu, en ce qui concerne la désignation des entités, la notion de *référence directe* grâce à une *variable*. Ceci correspond en gros à une boîte noire dans

laquelle on peut placer certaines entités, le *type* de la variable indiquant la taille des entités que l'on peut y placer.

L'**indexation indirecte** repose sur le principe suivant : on utilise une nouvelle sorte de variables, dites variables **pointeurs** ; le contenu d'une telle variable ne sera pas une entité du type voulu mais l'*adresse* de cette entité (dite entité **pointée**).

L'intérêt de cette façon de faire est que le contenu d'une telle variable pointeur a une taille fixe, quel que soit le type de l'entité pointée (qui peut être de taille très grande).

Types pointeurs.- À tout type correspond un type pointeur. L'intérêt de ceci est dû au fait que, si le contenu d'une variable pointeur, lui, est de taille constante, la taille de l'entité pointée, elle, dépend du type de cette entité ; lorsqu'on réserve l'emplacement pour l'entité pointée il faut donc en connaître la taille, indiquée par le type du type pointeur.

Déclaration d'une variable de type pointeur.- Une variable de type pointeur, ou plus concisément un **pointeur**, se déclare comme n'importe quelle variable.

Le problème de l'initialisation d'un pointeur.- Une variable de type simple s'initialise directement, soit par lecture, soit par affectation.

Il est dangereux d'attribuer directement une valeur à un pointeur car celle-ci est une adresse de la mémoire centrale, qui peut donc avoir été choisie par le compilateur pour une autre variable ou, pire, pour une partie du programme. D'ailleurs, dans beaucoup de langages (tel que le langage Pascal), l'adresse n'est pas accessible directement.

Accès au contenu pointé.- La référence à une variable pointeur donnera l'adresse de l'entité voulue et non l'entité elle-même. Il faut donc un moyen pour désigner cette entité. La façon de faire dépend beaucoup du langage considéré.

7.6.1.2 Mise en place en langage C

Type pointeur.- À chaque type τ est associé un type pointeur $\tau*$ correspondant, noté en faisant suivre le nom du premier type par une étoile.

Déclaration d'une variable d'un type pointeur.- La syntaxe de déclaration d'une variable de nom `ptr` (pour *PoinTeR* ou *PoinTeuR*) de type pointeur pointant vers une entité de type `type` est :

```
type *ptr;
```

en utilisant l'astérisque, qui est l'**opérateur de pointeur**.

On dit aussi que la variable `ptr` est de type `type*`.

Remarque.- Le type pointeur est `type*` mais l'astérisque se place traditionnellement devant la variable et non derrière le type, ce qui permet des déclarations du genre :

```
int *ptr, i ;
```

Attention ! On déclare dans ce cas en même temps une variable `i` de type entier et une variable `ptr` de type pointeur vers un entier et non deux variables pointeur.

Initialisation des pointeurs.- On peut initialiser une variable de type pointeur de plusieurs façons :

- 1^o) **affectation directe**, par exemple :

```
ptr = 24680;
```

évidemment à éviter, car on ne sait pas ce qui a été placé à cette adresse par le compilateur (sauf pour l'adresse fixe de certains périphériques).

- 2^o) affectation grâce à l'opérateur d'adressage, par exemple :

```
int i;
int *ptr;
ptr = &i;
```

- 3^o) affectation d'un autre pointeur, par exemple :

```
ptr1 = ptr2;
```

Accès au contenu pointé.- Pour accéder au contenu désigné par le pointeur `ptr` on utilise encore l'astérisque comme **opérateur d'indirection** (ou **opérateur de contenu** ou **opérateur de référence**). Le contenu désigné par `ptr` a pour nom (composé) `*ptr`.
On a par exemple :

```
int i;
int *ptr;
i = 3;
ptr = &i;
i = *ptr;
*ptr = 34;
```

7.6.2 Passage des arguments de fonctions par adresse

Introduction.- Nous l'avons déjà dit, et nous le répétons : la transmission par référence n'existe pas en langage C ; elle est émulée par le **passage par adresse**. Le principe en est le suivant :

On communique l'adresse d'une variable, ce qui permet d'en modifier le contenu (sans modifier l'adresse elle-même).

L'adresse d'une variable est manipulée par un pointeur. Les paramètres formels doivent être des pointeurs ; les paramètres effectifs doivent être des adresses, par exemple des noms de variables précédés de l'opérateur d'adressage `&`.

Exemple.- Rappelons que le programme suivant :

```
/* valeur_2.c */

#include <stdio.h>


void affiche(int n)
  {
  n = 4;
  printf("%d \n", n);
  }

void main(void)
  {
  int n = 3;
  affiche(n);
  printf("%d \n", n);
  }
```

avec passage des paramètres par valeur fait afficher 4 puis 3. Considérons le programme modifié suivant :

```
/* adresse.c */
#include <stdio.h>

void affiche(int *n)
     {
     *n = 4;
     printf("%d \n", *n);
     }

void main(void)
     {
     int n = 3;
     affiche(&n);
     printf("%d \n", n);
     }
```

Son exécution fait afficher 4 puis 4. La fonction `affiche()` a donc changé la valeur de `n` dans la fonction principale.

7.7 Philosophie du raffinement successif

Nous avons vu plusieurs raisons pour utiliser des fonctions, c'est-à-dire des sous-programmes. Une utilisation importante des sous-programmes conduit à la méthode de programmation connue sous le nom de **raffinement successif**.

7.7.1 Conception générale

Un bon plan d'attaque pour concevoir un algorithme est de décomposer la tâche à accomplir en quelques sous-tâches importantes, puis de décomposer chacune de ces sous-tâches en des sous-tâches plus petites, et ainsi de suite. À la fin les sous-tâches deviennent si petites qu'elles se programment très facilement, en langage C ou dans n'importe lequel des langages de programmation choisi.

On considère en général que le corps du programme principal ou d'une fonction ne devrait jamais dépasser une dizaine de lignes.

7.7.2 Exemple : opérations sur les rationnels

7.7.2.1 Le problème

Nous voulons concevoir un programme concernant les opérations sur les nombres rationnels, simulant une petite calculette interactive permettant d'effectuer les quatre opérations sur les nombres rationnels. On termine en proposant le pseudo-rationnel **0/0**. Un exemple de session d'utilisation de ce programme est le suivant, où nous écrivons en gras ce que l'on doit écrire au clavier (suivi d'un retour chariot après chaque entier et chaque signe d'opération) et de façon normale la réponse :

1/2 + 1/3 = 5/6
1/2 x 1/3 = 1/6

```
1/2 : 1/3 = 3/2
-1/2 + 1/3 = - 1/6
245/28 + 26/49 = 1819/196
258/-48 + 25/491 = 2797/11784
48/45 x 456/29 = 2432/145
0/0 bye
```

Ce programme ne peut être que de taille assez importante. Conformément à la philosophie précédente nous allons donc programmer petit à petit ce que nous désirons et en le concevant de la façon la plus modulaire possible de façon à pouvoir réutiliser des fonctions lorsqu'on passe d'un programme à l'autre.

7.7.2.2 Problème de la saisie d'un rationnel

Représentation des rationnels.- Le premier problème qui se pose est de représenter les rationnels. Un rationnel n'est rien d'autre que le quotient de deux entiers relatifs, le *numérateur* sur le *dénominateur*, le dénominateur étant non nul. Pour entrer un rationnel, on entrera d'abord un entier `a`, son numérateur par convention, puis un autre entier (non nul) `b`, son dénominateur. L'écho à l'écran devra être de la forme : `a/b`.

Il faut bien indiquer à un certain moment que nous avons fini d'entrer le premier entier. On le fait habituellement par un retour chariot.

Problème de la fonction de saisie.- On pourrait utiliser l'instruction (en fait la fonction) `scanf()`, mais à ce moment-là on va à la ligne, ce qui n'est pas très beau. On va donc devoir écrire nous-même une fonction de saisie d'un entier sans aller à la ligne.

En fait ceci n'est pas possible en langage C standard. Nous allons donc commencer par voir quelques compléments sur le traitement des caractères en langage C pour certains systèmes d'exploitation.

7.7.2.3 Compléments sur les caractères

Type caractère.- Nous avons déjà vu le type simple caractère `char` avec son format `%c` pour les entrées et les sorties formatées.

Constantes caractères.- Une constante caractère doit être entourée d'apostrophes verticales, à ne pas confondre avec les guillemets pour les constantes texte. Ce qui suit est un exemple de portion de programme correct :

```
char c;
c = 'a';
```

Constantes spéciales.- Certaines constantes caractère n'apparaissent pas sur le clavier : on dit qu'elles sont **non affichables**. On peut cependant les désigner par leur numéro de code précédé d'une contre-oblique. Par exemple, si le code utilisé est ASCII, on peut écrire :

```
char c, d;
c = '\0';
d = '\7';
```

si on veut initialiser au caractère nul et au bip respectivement.

Pour ce dernier on aurait aussi pu écrire :

```
d = '\a';
```

puisque nous avons déjà vu que cela correspondait à l'alarme. On pourra tester tout ceci grâce à un programme `char_1.c`.

Saisie et écriture des caractères.- On peut saisir et écrire un caractère seul grâce aux fonctions prédéfinies `getchar()` et `putchar()`. La première fonction est sans argument et renvoie un entier, le code du caractère en question ; elle est bloquante et attend un caractère suivi d'un retour chariot. La seconde a un argument entier et affiche le caractère dont le code est cet entier.

Voici un exemple d'utilisation :

```
/* char_2.c */

#include <stdio.h>

void main(void)
     {
     char c;
     printf("Entrez un caractere : ");
     c = getchar();
     putchar(c);
     putchar(c);
     putchar('\n');
     }
```

qui permet d'entrer un caractère et de l'afficher deux fois (de façon accolée) sur la ligne suivante.

Inconvénient.- L'inconvénient de la fonction `getchar()` est qu'il faut terminer par un retour chariot, ce qui est déjà gênant en soi (il faut appuyer sur deux touches) mais de plus ce passage à la ligne est pris en compte en écho à l'écran.

Ce problème ne semble pas avoir été pris en compte dans la définition du langage C. Il l'est dans le cas des compilateurs pour MS-DOS (et Windows) en ajoutant des fonctionnalités.

7.7.2.4 Cas des compilateurs pour MS-DOS

Nouvelles fonctions.- Dans le cas des compilateurs pour le système d'exploitation MS-DOS, les fonctions `getch()` et `getche()` permettent de résoudre l'inconvénient dont nous venons de parler. Elles permettent de saisir un caractère au clavier (sans écho ou avec écho à l'écran, respectivement), sans avoir à terminer par un retour chariot ; de plus on ne passe pas à la ligne à l'écran si on a choisi un écho. Les déclarations de ces fonctions se trouvent dans le fichier en-tête `conio.h` (évidemment pour entrée-sortie sur console). Les fonctions `putch()` et `putche()` existent également pour en faire le pendant mais elles correspondent toutes les deux à `putchar()`.

Exemple d'utilisation.- Le programme suivant demande deux caractères qu'il affiche dans l'ordre inverse, et tout cela sur une seule ligne :

```
/* char_3.c
 * non portable
 * uniquement pour MS-DOS */

#include <stdio.h>
#include <conio.h> /* non portable */
```

```
void main(void)
     {
     char c, d;
     printf("Entrez deux caracteres : ");
     c = getche(); /* non portable */
     d = getche(); /* non portable */
     putchar(d);
     putchar(c);
     putchar('\n');
     putchar('\n');
     }
```

Remarque importante.- Les fonctions `getch()` et `getche()` saisissent le passage à la ligne ('\n') comme un retour chariot ('\r').

7.7.2.5 Saisie d'un entier naturel

Introduction.- Il est facile de saisir un entier, avec écho à l'écran se terminant par un passage à la ligne, grâce à la fonction `scanf()`. Rien n'est prévu en langage C si on ne veut pas aller à la ligne. Nous allons concevoir une fonction `lire_entier()` qui effectue ce que nous voulons.

Principe.- L'idée est simple : on lit l'entier caractère par caractère (on dit au vol), les caractères étant des chiffres décimaux, en terminant la lecture par un retour chariot (à ne pas répercuter à l'écran).

Quel est le rapport entre ce qui est lu et l'entier voulu ? Considérons, par exemple, l'entier : **123**.

Cela signifie qu'on lit les caractères '**1**', '**2**' et '**3**' et que l'on veut l'entier :

$$n = 1 \times 100 + 2 \times 10 + 3.$$

Comment savoir, lorsqu'on lit le premier caractère, qu'il faut le multiplier, dans notre exemple, par **100** ?

On n'en a évidemment aucune idée. Heureusement la **méthode de Hörner** permet de ne pas s'en préoccuper. Il suffit d'écrire :

$$n = (1 \times 10 + 2) \times 10 + 3,$$

ou, plus exactement, pour ne pas s'occuper des parenthèses :

$$n = 0,$$

$$n = n \times 10 + 1,$$

$$n = n \times 10 + 2,$$

$$n = n \times 10 + 3.$$

Une fonction.- Ceci nous permet d'écrire la fonction **lire_entier()** que nous incorporons, comme d'habitude, dans un programme complet, pour la tester.

```
/* entier.c */

#include <stdio.h>
#include <conio.h> /* non portable */

void lire_entier(int *n)
     {
     char c;
     *n = 0;
     c = getch();
     while (c != '\r')
        {
        if (c == '0') *n = (*n)*10;
        if (c == '1') *n = (*n)*10 + 1;
        if (c == '2') *n = (*n)*10 + 2;
        if (c == '3') *n = (*n)*10 + 3;
        if (c == '4') *n = (*n)*10 + 4;
        if (c == '5') *n = (*n)*10 + 5;
        if (c == '6') *n = (*n)*10 + 6;
        if (c == '7') *n = (*n)*10 + 7;
        if (c == '8') *n = (*n)*10 + 8;
        if (c == '9') *n = (*n)*10 + 9;
        putchar(c);
        c = getch();
        }
    } /* fin de lire_entier */

void main(void)
     {
     int n;
     clrscr(); /* non portable */
     printf("Entrer un entier : ");
     lire_entier(&n);
     printf(".\n");
     printf("Cet entier est : %d. \n", N);
     }
```

Remarque.- Nous avons également utilisé la fonction `clrscr()` de `conio.h` permettant d'effacer l'écran (évidemment pour *CLeaR SCReen*).

7.7.2.6 Saisie d'un entier relatif

Introduction.- On considère qu'un *entier relatif* est un entier naturel précédé (sans espace) soit du signe '-', soit du signe '+', soit du mot vide. Si nous étions sûrs d'avoir un signe on pourrait réutiliser le programme précédent ; comme nous n'en sommes pas sûrs il faut adapter ce programme.

Une fonction.- On peut utiliser la fonction `lire_relatif()` du programme suivant :

```
/* relatif.c */
```

```
#include <stdio.h>
#include <conio.h> /* non portable */

void lire_relatif(int *n)
     {
     char c, sign;
     *n = 0;
     c = getch();
     if ((c == '-') || (c == '+'))
        {
        sign = c;
        putchar(c);
        c = getch();
        }
     else sign = '+';
     while (c != '\r')
        {
        if (c == '0') *n = (*n)*10;
        if (c == '1') *n = (*n)*10 + 1;
        if (c == '2') *n = (*n)*10 + 2;
        if (c == '3') *n = (*n)*10 + 3;
        if (c == '4') *n = (*n)*10 + 4;
        if (c == '5') *n = (*n)*10 + 5;
        if (c == '6') *n = (*n)*10 + 6;
        if (c == '7') *n = (*n)*10 + 7;
        if (c == '8') *n = (*n)*10 + 8;
        if (c == '9') *n = (*n)*10 + 9;
        putchar(c);
        c = getch();
        }
    if (sign == '-') *n = - (*n);
    } /* fin de lire_relatif */

void main(void)
     {
     int n;
     clrscr(); /* non portable */
     printf("Entrer un entier relatif : ");
     lire_relatif(&n);
     printf(".\n");
     printf("Cet entier relatif est : %d.\n",N);
     }
```

7.7.2.7 Saisie d'un rationnel

La fonction `lire_rationnel()` n'est pas difficile à écrire à partir de ce que nous venons de faire. Nous avons, par exemple :

```
/* rationnel.c */
```

```
#include <stdio.h>
#include <conio.h> /* non portable */

void lire_relatif(int *n);

void lire_rationnel(int *a, int *b)
     {
     lire_relatif(a);
     printf("/");
     lire_relatif(b);
     } /* fin de lire_rationnel */

void main(void)
     {
     int a, b;
     printf("Entrez un rationnel, d\'abord le \n ");
     printf("numerateur puis le denominateur : ");
     lire_rationnel(&a, &b);
     printf(".\n");
     printf("Ce rationnel est %d/%d.\n", a, b);
     }
```

7.7.2.8 Affichage d'un rationnel

Forme canonique d'un rationnel.- Un rationnel ne s'écrit pas de façon unique sous la forme **a/b**. Par exemple **4/6** est aussi égal à **6/9**, mais surtout à **2/3**, qui est sa *forme canonique*. Les résultats devront être écrits sous forme canonique : le signe, s'il existe, doit concerner le numérateur ; le numérateur et le dénominateur doivent être divisés par leur pgcd.

Un programme.- Ceci nous conduit à la fonction `simplifie()` suivante que l'on peut insérer dans un petit programme qui demande un rationnel et affiche sa forme canonique :

```
/* affiche.c */

#include <stdio.h>
#include <conio.h> /* non portable */

int pgcd(int a, int b);

void simplifie(int *a, int *b)
     {
     int d;
     d = pgcd(*a,*b);
     *a = (*a)/d;
     *b = (*b)/d;
     if (*b < 0)
        {
        *a = - (*a);
        *b = - (*b);
```

```
        }
     } /* fin de simplifie() */

void lire_relatif(int *k);

void lire_rationnel(int *a, int *b);

void main(void)
     {
     int a, b;
     printf("Entrer un rationnel : ");
     lire_rationnel(&a, &b);
     simplifie(&a, &b);
     printf(" = %d/%d.\n", a, b);
     }
```

dans lequel nous vous laissons le soin de reporter le code pour `pgcd()`, `lire_relatif()` et `lire_rationnel()`.

7.7.2.9 Addition des rationnels

Introduction.- La programmation de l'addition des rationnels est facile en utilisant la formule :

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}.$$

Il faut juste penser à ne pas oublier de simplifier le résultat obtenu.

Un programme.- Ceci nous conduit, par exemple, au programme suivant :

```
/* addition.c */

#include <stdio.h>
#include <conio.h> /* non portable */

int pgcd(int a, int b);
void simplifie(int *a, int *b);
void lire_relatif(int *k);
void lire_rationnel(int *a, int *b);

void add_rat(int a, int b, int c, int d, int *e, int *f)
     {
     *e = a*d + b*c;
     *f = b*d;
     simplifie(e,f);
     } /* fin de add_rat() */

void main(void)
     {
     int a,b,c,d,e,f;
     lire_rationnel(&a, &b);
```

```
     printf(" + ");
     lire_rationnel(&c, &d);
     add_rat(a,b,c,d,&e,&f);
     printf(" = %d/%d.\n", e, f);
     }
```

7.7.2.10 Soustraction, multiplication et division des rationnels

Introduction.- Ces opérations se traitent de façon tout à fait analogue en utilisant les formules :

$$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd},\ \frac{a}{b}.\frac{c}{d} = \frac{ac}{bd},\ \frac{a}{b} : \frac{c}{d} = \frac{ad}{bc}.$$

Un programme.- Contentons-nous de donner les fonctions :

```
void sub_rat(int a, int b, int c, int d, int *e, int *f)
     {
     *e = a*d - b*c;
     *f = b*d;
     simplifie(e,f);
     } /* fin de sub_rat() */

void mult_rat(int a, int b, int c, int d, int *e, int *f)
     {
     *e = a*c;
     *f = b*d;
     simplifie(e,f);
     } /* fin de mult_rat() */

void div_rat(int a, int b, int c, int d, int *e, int *f)
     {
     *e = a*d;
     *f = b*c;
     simplifie(e,f);
     } /* fin de div_rat() */
```

7.7.2.11 Un programme complet

Nous pouvons maintenant nous intéresser au programme complet, en portant tous nos efforts sur le programme principal.

```
/* op_rat.c */

#include <stdio.h>
#include <conio.h> /* non portable */

int pgcd(int a, int b);
void simplifie(int *a, int *b);
void lire_relatif(int *k);
void lire_rationnel(int *a, int *b);
void add_rat(int a, int b, int c, int d, int *e, int *f);
```

```
void sub_rat(int a, int b, int c, int d, int *e, int *f);
void mult_rat(int a, int b, int c, int d, int *e, int *f);
void div_rat(int a, int b, int c, int d, int *e, int *f);

void main(void)
     {
     int a,b,c,d,e,f;
     char op;
     lire_rationnel(&a, &b);
     while ((a != 0) || (b != 0))
        {
        printf(" ");
        op = getche();
        printf(" ");
        lire_rationnel(&c, &d);
        if (op == '+') add_rat(a,b,c,d,&e,&f);
        if (op == '-') sub_rat(a,b,c,d,&e,&f);
        if (op == 'x') mult_rat(a,b,c,d,&e,&f);
        if (op == ':') div_rat(a,b,c,d,&e,&f);
        printf(" = %d/%d,\n",e,f);
        lire_rationnel(&a,&b);
        }
     printf(" bye\n");
     }
```

7.7.2.12 Cas des compilateurs sous Unix

Introduction.- Pour réaliser le programme précédent sous MS-DOS, nous avons utilisé le fichier en-tête `conio.h` et, en particulier, les fonctions `getch()` et `getche()`. Ces fonctions ne sont pas implémentées pour les compilateurs sous Unix. On peut les implémenter grâce, par exemple, au programme suivant mais on remarquera que nous sommes obligés de faire de la programmation système.

Implémentation de getch().- Il suffit d'implémenter `getch()` puisque `getche()` n'est rien d'autre que la fonction `getch()` suivie d'un `putchar()`.

```
/* unix.c */

#include <stdio.h>
#include <termios.h> /* non portable, Unix */

int getch(void)
    {
    struct termios initial, new;
    char c;
    tcgetattr(fileno(stdin), &initial);
    new = initial;
    new.c_lflag &= ~ICANON;
    new.c_lflag &= ~ECHO;
    tcsetattr(fileno(stdin), TCSAFLUSH, &new);
```

```
    c = getchar();
    tcsetattr(fileno(stdin), TCSANOW, &initial);
    return (unsigned char) c;
    }

void main(void)
     {
     char c, d;
     printf("Entrez deux caracteres : ");
     c = getch();
     d = getch();
     putchar(d);
     putchar(c);
     putchar('\n');
     putchar('\n');
     }
```

Commentaires.- Nous n'allons pas entrer dans tous les détails. Ceci sera éventuellement revu dans le cours de programmation système.

- 1^o) On fait appel à une bibliothèque (`termios.h`) de programmation système concernant les terminaux, conforme au standard POSIX.

- 2^o) On utilise des structures (à savoir `initial` et `new`) de type `termios`. La notion de structure est un outil de programmation générale que nous verrons plus tard.

- 3^o) Le fichier `stdin` est le périphérique d'entrée standard, notre clavier. Il porte un numéro que l'on récupère grâce à la fonction `fileno()`.

- 4^o) La fonction `tcgetattr()` (pour *Terminal Configuration GET ATTRibutes*) permet d'obtenir la configuration actuelle du clavier et de la sauvegarder dans la structure `initial`.

- 5^o) On commence alors à reconfigurer le clavier en commençant par recopier la configuration initiale et on change que les deux points qui nous intéressent : la première ligne indique que nous ne voulons pas être dans le mode d'entrée canonique ; la seconde ligne indique que nous ne voulons pas d'écho (à l'écran).

- 6^o) Nous attribuons alors cette configuration au clavier grâce à la fonction `tcsetattr()`.

- 7^o) Il ne faut pas oublier de reconfigurer le clavier tel qu'il était auparavant lorsqu'on sort du programme.

7.8 Réutilisation : calcul des nombres harmoniques

Introduction.- Nous allons montrer un autre intérêt de la programmation modulaire, à savoir que l'on peut **réutiliser** des parties de programmes, c'est-à-dire des sous-programmes.

Les nombres harmoniques.- Pour un entier naturel non nul n, on appelle n-ième **nombre harmonique**, et on note H_n, la somme des inverses des entiers de 1 à n :

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}.$$

Un programme.- Nous allons concevoir un programme qui calcule H_n.

```
/* harmonic.c
 * Ce programme calcule le n-ieme
 * nombre harmonique
 * sous MS-DOS
 */

#include <stdio.h>
#include <conio.h> /* non portable */

void lire_entier(int *a);
int pgcd(int a, int b);
void simplifie(int *a, int *b);
void add_rat(int a, int b, int c, int d, int *e, int *f);

void main(void)
     {
     int n, a, b, i;
     clrscr(); /* non portable */
     printf("H(");
     lire_entier(&n);
     a = 1;
     b = 1;
     for (i = 2; i <= n; i++) add_rat(a,b,1,i,&a,&b);
     printf(") = %d/%d.\n", a, b);
     }
```

On obtient, par exemple :

H(7) = 363/140.

Exercices

Exercice 1.- Dans le programme `entier.c`, on a dix tests pour déterminer le chiffre. Dans la plupart des codes (c'est le cas du code ASCII), les chiffres ont des codes successifs. On utilise donc l'astuce suivante :

```
chiffre = C - '0';
```

qui donne la valeur du chiffre.

Écrire un programme C effectuant la même chose que le programme `entier.c` *en utilisant l'astuce précédente.*